Enrolment No./Seat No_____

# GUJARAT TECHNOLOGICAL UNIVERSITY

**BE - SEMESTER–IV EXAMINATION – SUMMER 2025**

**Subject Code:3140705 Date:23-05-2025**

**Subject Name: Object Oriented Programming -I**

**Time: 10:30 AM TO 01:00 PM Total Marks:70**

**Instructions:**

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

| | | | MARKS |
|---|---|---|---|
| **Q.1** | **(a)** | Explain method overriding and method overloading in Java. | **03** |
| | **(b)** | Demonstrates use of BufferedReader and the readLine() method. | **04** |
| | **(c)** | Define and explain object and class in Java with appropriate example. | **07** |
| **Q.2** | **(a)** | What is constructor? What is it role? Explain various features/characteristics of constructors. | **03** |
| | **(b)** | What are different types of access modifier? | **04** |
| | **(c)** | What is purpose of using methods? How do you declare a method? How do you invoke a method? | **07** |
| | | **OR** | |
| | **(c)** | What is difference between final, finally and finalize. | **07** |
| **Q.3** | **(a)** | What is super class? | **03** |
| | **(b)** | Define interface in java. | **04** |
| | **(c)** | Explain inheritance with example. | **07** |
| | | **OR** | |
| **Q.3** | **(a)** | What is use of super keyword? | **03** |
| | **(b)** | What is package concept and Describe the use of package. | **04** |
| | **(c)** | Explain polymorphism with example. | **07** |
| **Q.4** | **(a)** | Explain use of throw in exception handling with example. | **03** |
| | **(b)** | Explain creation of different shapes in JAVAFX application? | **04** |
| | **(c)** | Explain Generics classes with example. | **07** |
| | | **OR** | |
| **Q.4** | **(a)** | Explain difference between throw and throws. | **03** |
| | **(b)** | Write programs to deal with MouseEvents. | **04** |
| | **(c)** | Explain Generics methods with example. | **07** |
| **Q.5** | **(a)** | Demonstrate use of the Animation, PathTransition. | **03** |
| | **(b)** | Describe the life cycle of a thread object. | **04** |
| | **(c)** | Explain use of Linked List collection class with example. | **07** |
| | | **OR** | |
| **Q.5** | **(a)** | Create a radio button using the RadioButton class and group radio buttons using a ToggleGroup. | **03** |
| | **(b)** | Explain runnable interface. | **04** |
| | **(c)** | Explain Sets with examples. | **07** |

**********

Object Oriented Programming –I 2025

Q.1 (a) Explain method overriding and method overloading in Java.

# Q.1 (a) Method Overriding and Method Overloading in Java

## 1. Method Overloading

### Definition

Method overloading in Java means **creating multiple methods in the same class with the same name but different parameters**.
The difference can be in:

- Number of parameters
- Type of parameters
- Order of parameters

This provides **compile-time polymorphism**, because the method is selected during **compilation**.

### Key Characteristics

- Occurs **within the same class**.
- Method name is same but **signature is different**.
- Return type may be same or different.
- Provides **flexibility**, allowing developers to use the same method name for similar tasks.
- Also known as **static polymorphism** or **early binding**.

### Example of Method Overloading

```
class MathOperation {
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with 3 parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method with float parameters
    float add(float a, float b) {
        return a + b;
    }
```

```
}
```

### Explanation

Here, the method name `add()` is same, but:

- First has 2 integers
- Second has 3 integers
- Third has 2 floats

Thus, compiler decides which method to call based on arguments.

# 2. Method Overriding

### Definition

Method overriding in Java happens when a **subclass provides its own implementation** of a method that is already defined in its **parent class**.
The method in the child class must have:

- Same name
- Same return type
- Same parameter list

This supports **runtime polymorphism**, because method selection happens **at runtime**.

### Key Characteristics

- Requires **inheritance** (two classes).
- Method signature must be **exactly same**.
- Access modifier cannot be more restrictive than the parent method.
- Achieves **dynamic polymorphism** or **late binding**.
- Used to change/extend parent class behavior.

### Example of Method Overriding

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

(b) Demonstrates use of BufferedReader and the readLine() method.

In Java, **BufferedReader** is a class used to read text efficiently from input sources such as the keyboard, files, or network streams.
It reads data in a **buffer**, which makes input faster compared to using low-level classes like `InputStreamReader` or `FileReader`.

One of the most commonly used methods of BufferedReader is **readLine()**, which reads a **complete line of text** at a time.

---

# Explanation of BufferedReader

- `BufferedReader` belongs to the package **java.io**.
- It adds buffering capability to improve performance.
- It is commonly used with `InputStreamReader` when reading data from the keyboard.

### Syntax

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

Here:

- `System.in` → input from keyboard
- `InputStreamReader` → converts byte stream to character stream
- `BufferedReader` → reads the character stream efficiently

---

# Explanation of readLine() Method

- `readLine()` reads **one entire line** until the user presses Enter.
- It returns the line as a **String**.
- If the end of the stream is reached, it returns **null**.

### Syntax

```
String line = br.readLine();
```

---

# Program to Demonstrate BufferedReader and readLine()

### Example Program

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class ReadExample {
    public static void main(String[] args) throws IOException {
```

```
        // Creating BufferedReader object
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

        System.out.println("Enter your name:");

        // Reading one line using readLine()
        String name = br.readLine();

        System.out.println("You entered: " + name);
    }
}
```

---

## Advantages of Using BufferedReader

- Faster input due to internal buffering
- Can read large text efficiently
- Supports reading full lines using `readLine()`
- Suitable for handling file input operations as well

(c) Define and explain object and class in Java with appropriate example

# 1. Class in Java

## Definition

A **class** in Java is a **blueprint**, **template**, or **model** from which objects are created.
It defines:

- Data members (variables)
- Methods (functions)
- Constructors
- Behaviors and properties

A class does **not occupy memory** until an object is created from it.

---

## Explanation

A class represents a **group of similar objects**.
It bundles **data** and **methods** into a single unit following OOP principles such as **encapsulation**.

A class can contain:

- Fields / variables
- Methods
- Constructor
- Blocks
- Nested classes

### Syntax of a Class

```
class ClassName {
    // data members
    // methods
}
```

---

# 2. Object in Java

## Definition

An **object** is an **instance of a class**.
It represents a real-world entity such as a student, car, mobile, employee, etc.

An object has:

- **State** (data/attributes)
- **Behavior** (methods)
- **Identity** (unique existence in memory)

Objects occupy **memory** and allow us to access the class's fields and methods.

---

## Explanation

Once a class is created, no memory is allocated until an object is created using the `new` keyword.

### Syntax of Object Creation

```
ClassName obj = new ClassName();
```

---

# 3. Example Demonstrating Class and Object

```
class Student {
    // data members (state)
    String name;
```

```
    int age;

    // method (behavior)
    void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class Demo {
    public static void main(String[] args) {

        // Creating object of Student class
        Student s1 = new Student();

        // Assigning values
        s1.name = "Komal";
        s1.age = 20;

        // Calling method using object
        s1.display();
    }
}
```

---

Q.2 (a) What is constructor? What is it role? Explain various features/characteristics of constructors.

# What is a Constructor?

A **constructor** in Java is a **special method** that is automatically called when an object is created.
Its name is **same as the class name**, and it has **no return type**, not even `void`.

A constructor is used to **initialize the object**, i.e., to assign initial values to the object's data members when the object is created.

## Role of Constructor

The role of a constructor includes:

1. **Object Initialization:**
   Assigning initial values to the object's variables.
2. **Setting Up Resources:**
   Preparing memory, connections, or states needed before using the object.
3. **Ensuring Valid Object State:**
   Making sure every object begins with meaningful or valid data.
4. **Supporting Overloading:**
   Providing different ways to create objects with different initial values.

5. **Handling Default Behavior:**
   If no constructor is defined, Java automatically provides a default constructor, ensuring object creation is always possible.

# Features / Characteristics of Constructors

## 1. Same Name as the Class

A constructor must have the exact same name as the class. This helps Java identify it as a constructor and not a normal method.

---

## 2. No Return Type

Constructors do not have any return type—not even `void`.
They implicitly return the newly created object.

---

## 3. Automatically Invoked

Constructors are called automatically at the moment an object is created using the `new` keyword.
Programmers do not call them manually.

---

## 4. Constructors Can Be Overloaded

A class can have more than one constructor with different parameter lists.
This allows creating objects in multiple ways depending on the requirement.

---

## 5. Constructors Are Not Inherited

A child class does not inherit its parent's constructors.
However, the child can call a parent constructor using the `super()` keyword.

---

## 6. Used Primarily for Initialization

Constructors are designed specially to initialize data members of a class, ensuring each object starts with valid values.

---

(b) What are different types of access modifier?

Access modifiers in Java are **keywords** used to control the **visibility** and **accessibility** of classes, methods, variables, and constructors.
They help implement **encapsulation**, one of the key principles of object-oriented programming.

Java provides **four** types of access modifiers.

---

# 1. Public

- The `public` modifier gives the **widest access level**.
- A public class, method, or variable can be accessed **from anywhere** in the program:
  - within the same class
  - within the same package
  - from other packages
  - from subclasses and non-subclasses
- Used when we want to provide **full access**.

---

# 2. Private

- The `private` modifier gives the **most restricted** access.
- A private variable or method can be accessed **only within the same class**.
- It is **not accessible** in:
  - other classes
  - child classes
  - other packages
- It is mainly used to **protect data** and **encapsulate sensitive information**.

---

# 3. Protected

- The `protected` modifier allows access:
  - within the same class
  - within the same package
  - and in **subclasses**, even if they are in different packages
- It is commonly used when we want to **share data with child classes** but restrict access from unrelated classes.

---

# 4. Default (No Modifier)

- Also called **package-private**.
- When no access modifier is written, Java provides **default** access.
- Members are accessible:
  - within the same class
  - within the same package
- Not accessible from classes in other packages.
- Useful when we want to allow package-level access but not public access.

(c) What is purpose of using methods? How do you declare a method? How do you invoke a method?

# Purpose of Using Methods

Methods in Java are used to **define reusable blocks of code** that perform specific tasks. They help in writing clean, organized, and modular programs.

### Main purposes:

### 1. Code Reusability

A method can be written once and used multiple times, reducing repetition.

### 2. Better Organization and Readability

Dividing large programs into smaller methods makes the code easier to read, understand, and manage.

### 3. Modularity

Methods allow splitting a complex problem into smaller subproblems, improving structure and making debugging easier.

### 4. Easy Maintenance

If changes are needed, they can be made inside a method without affecting the entire program.

### 5. Avoiding Duplication

Common logic can be placed inside a method and reused, which reduces errors and improves consistency.

### 6. Encapsulation

Methods help hide internal details and expose only required functionality.

### How Do You Declare a Method?

A method is **declared inside a class** using the following syntax:

### General Syntax

```
returnType methodName(parameterList) {
    // method body
}
```

### Explanation

- **returnType:** The type of value returned (`int`, `void`, `String`, etc.)
- **methodName:** Name of the method
- **parameterList:** Values passed to the method (optional)
- **method body:** Statements that define what the method does

### Example of Method Declaration

```
int add(int a, int b) {
    return a + b;
}
```

# How Do You Invoke (Call) a Method?

To execute a method, we **call** or **invoke** it.

### Invocation Syntax

```
objectName.methodName(arguments);
```

- For **non-static** methods → call using object
- For **static** methods → call using class name or directly

### Example

```
MyClass obj = new MyClass();  // create object
obj.show();                   // invoking non-static method

MyClass.display();            // invoking static method
```

# Example:

```
class Calculator {



   // method declaration
```

```
    int add(int x, int y) {
        return x + y;
    }

    // static method declaration
    static void greet() {
        System.out.println("Welcome to Calculator!");
    }
}

public class Demo {
    public static void main(String[] args) {

        // invoking static method
        Calculator.greet();

        // creating object
        Calculator c = new Calculator();

        // invoking non-static method
        int result = c.add(10, 20);

        System.out.println("Addition: " + result);
    }
}
```

OR:

(c) What is difference between final, finally and finalize.

# 1. final (Keyword)

The **final** keyword is used to apply restrictions on classes, methods, and variables.
It tells the compiler that something **cannot be changed** or **modified** later.

### Uses of final:

1. **final variable:**
   A final variable becomes a constant. Its value cannot be changed after initialization.
2. **final method:**
   A final method cannot be overridden by any subclass.
   This ensures the method's behavior remains the same.
3. **final class:**
   A final class cannot be inherited.
   This is used for security reasons or when the class design should remain unchanged.

Thus, **final** is mainly used to create **constants**, avoid **method overriding**, and stop **inheritance**.

---

# 2. finally (Block)

The **finally** block is used in **exception handling** with try-catch.

Its main purpose is to execute certain statements **whether an exception occurs or not**.
It is used for **cleanup operations** like:

- closing files
- releasing database connections
- freeing resources
- printing completion messages

The finally block always runs, even if:

- an exception occurs
- no exception occurs
- a return statement is used inside try/catch

Therefore, **finally** ensures that the program does not leave important tasks unfinished.

---

# 3. finalize() (Method)

`finalize()` is a **method** defined in the `Object` class.
It is called by the **Garbage Collector** before destroying an object from memory.

Purpose of finalize():

- to perform cleanup just before the object is removed
- to release resources that the object was using
- to give a last chance to perform important operations before destruction

However, in modern Java, finalize() is rarely used because:

- it is not guaranteed when or if it will run
- better mechanisms like try-with-resources exist
- Java 9 and later versions mark it as deprecated

### ✅ 1. final

```
final int speed = 60;   // 'speed' cannot be changed after this
```

---

### ✅ 2. finally

```
try {
int x = 5/0;
}
finally { System.out.println("Cleanup done"); }
```

---

### ✅ 3. finalize()

```
protected void finalize()
 {
System.out.println("Object destroyed"); }
```

## Q.3 (a) What is super class?

A **superclass** in Java is the **parent class** from which another class (called the **subclass**) *inherits* properties and behaviors.
It provides **common variables and methods** that can be reused by all its subclasses.
Inheritance helps in reducing code duplication and improves code reusability.

### ⋆ Key Points about Superclass

- It is also called **base class** or **parent class**.
- A subclass uses the **extends** keyword to inherit a superclass.
- Methods and variables defined in the superclass become available to subclasses.
- A subclass can also **override** superclass methods.

---

# ✅ Example

```
// Superclass (Parent Class)
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Subclass (Child Class)
class Dog extends Animal {
    void show() {
        System.out.println("Dog is an animal");
    }
}

public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound(); // Inherited from superclass
        d.show();  // Own method
    }
}
```

### ⭑ Explanation:

- **Animal** is the *superclass*.
- **Dog** is the *subclass* that extends Animal.
- The subclass inherits the **sound()** method from the superclass.

## (b) Define interface in java.

An **interface** in Java is a **reference type**, similar to a class, that is used to define a **set of abstract methods** and **constants** that a class must implement.
It acts as a **contract** or **blueprint** specifying *what a class should do*, but not *how it should do it*.
By implementing an interface, a class agrees to provide concrete definitions for all of the interface's abstract methods.

---

# Key Features / Characteristics of an Interface

1. **Abstract Methods**

    - All methods in an interface are abstract by default (i.e., they have no body).
    - From Java 8 onward, interfaces can also have **default** and **static** methods with a body.
2. **Constants**
    - Variables declared inside an interface are automatically **public, static, and final**.
    - They cannot be changed once initialized.
3. **Access Modifier**
    - All methods in an interface are implicitly **public**.
    - Interfaces themselves are also usually declared public.
4. **Implementation**
    - A class uses the **implements** keyword to implement an interface.
    - The class must provide concrete definitions for all abstract methods.
5. **Supports Multiple Inheritance**
    - Java does not support multiple inheritance with classes, but a class can implement **multiple interfaces**, enabling multiple inheritance of type.
6. **Cannot Instantiate**
    - You cannot create an object of an interface directly.
    - Only classes that implement the interface can be instantiated.

---

# Simple Example

```
// Interface declaration
interface Animal {
    void sound();   // abstract method
}

// Class implementing interface
class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}

public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();  // Output: Dog barks
    }
}
```

### Explanation

- `Animal` is the **interface** specifying the method `sound()`.
- `Dog` is the **class** that implements the interface and provides the method definition.
- This shows how an interface enforces a contract while allowing flexibility in implementation.

(c) Explain inheritance with example.

# Definition of Inheritance

Inheritance in Java is an **Object-Oriented Programming (OOP) concept** that allows a **class (child/subclass)** to acquire the **properties (fields) and behaviors (methods)** of another class (parent/superclass).

It is used to:

- **Reuse existing code**
- **Establish relationships** between classes
- **Support hierarchical classification**
- **Enable polymorphism**

---

# Key Features of Inheritance

1. **Code Reusability:**
   Methods and variables of the parent class are automatically available to the child class.
2. **Hierarchical Classification:**
   Organizes classes in a hierarchy from general to specific.
3. **Supports Polymorphism:**
   Overridden methods in child classes allow dynamic method dispatch.
4. **Reduces Redundancy:**
   Common code can be written in the parent class and reused in multiple child classes.
5. **Types of Inheritance in Java:**
   - **Single Inheritance:** One parent, one child
   - **Multilevel Inheritance:** Class inherits from a parent, and another class inherits from this child
   - **Hierarchical Inheritance:** Multiple classes inherit from one parent class
   - **Multiple Inheritance:** Java **does not support multiple inheritance with classes** (can be achieved with interfaces)

---

# Syntax of Inheritance

```
class ParentClass {
    // members
}

class ChildClass extends ParentClass {
    // additional members
}
```

- The keyword **extends** is used for class inheritance.
- The child class can **use or override** parent class members.

# Example of Inheritance

```
// Parent (Super) Class
class Animal {
    void eat() {
        System.out.println("Animal eats food");
    }
}

// Child (Sub) Class
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}

// Main Class
public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();   // inherited from Animal
        d.bark();  // own method of Dog
    }
}
```

### Explanation of Example

- **Animal** is the **superclass** (parent).
- **Dog** is the **subclass** (child) that extends Animal.
- The subclass **inherits the `eat()` method** from Animal.
- The subclass also has its own method `bark()`.
- Output:

```
Output:
Animal eats food
Dog barks
```

# Advantages of Inheritance

1. Promotes **code reusability**.
2. Simplifies **program maintenance**.
3. Enables **polymorphism** and method overriding.
4. Provides a **hierarchical class structure**.

OR

Q.3 (a) What is use of super keyword?

In Java, the **`super` keyword** is a reference variable used by a **subclass** to access members (variables, methods, or constructors) of its **immediate parent class (superclass)**.

It helps in **avoiding ambiguity** when a subclass has members with the same name as the superclass, and allows the subclass to **reuse parent class features**.

---

# Uses of super Keyword

## 1. Accessing Parent Class Variables

- If a **subclass variable has the same name** as a variable in the superclass, `super` can be used to refer to the **parent class variable**.

## 2. Calling Parent Class Methods

- When a **method is overridden** in the subclass, `super` can call the **parent class version** of the method.

## 3. Invoking Parent Class Constructor

- `super()` can be used in a subclass constructor to **call the constructor of the parent class**.
- It must be the **first statement** in the subclass constructor.
- This is useful to **initialize inherited variables** or perform setup defined in the superclass.

Example:

```
class Parent {
    int x = 10;
}
class Child extends Parent {
    int x = 20;
    void show() {
        System.out.println("Parent x = " + super.x);  // access parent variable
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Child c = new Child();
        c.show();
    }
}
```

**Output:**

```
Parent x = 10
```

(b) What is package concept and Describe the use of package.

# Definition of Package

In Java, a **package** is a **collection of related classes, interfaces, and sub-packages** grouped together under a single name.
It is similar to a **folder** in an operating system that contains multiple files.

Packages help organize code and **avoid naming conflicts** in large programs.

---

# Types of Packages in Java

1. **Built-in Packages**
   - Java provides many **predefined packages** for developers.
   - Examples:
     - `java.lang` → contains fundamental classes like `String`, `Math`, `Object`
     - `java.util` → contains utility classes like `ArrayList`, `Scanner`
     - `java.io` → contains classes for input/output operations
2. **User-defined Packages**
   - Developers can create their **own packages** to organize classes for a specific application.
   - Example: creating `package mypackage;` and storing related classes inside it.

---

# Use / Advantages of Packages

1. **Code Organization**
   - Groups related classes and interfaces together, making programs **organized and manageable**.
2. **Avoid Naming Conflicts**
   - Classes with the same name can exist in **different packages** without conflict.
3. **Code Reusability**
   - Classes in a package can be **imported and reused** in other programs.
4. **Access Control**
   - Packages allow **controlled access** to classes, methods, and variables using **access modifiers** like `public` and default (package-private).
5. **Simplifies Maintenance**
   - Updating, debugging, and maintaining code becomes easier when classes are logically grouped.
6. **Encapsulation Support**
   - Packages can hide internal implementation details and expose only necessary interfaces.

---

# Example of Using a Package

### Creating a Package

```
package mypackage;

public class Hello {
    public void show() {
        System.out.println("Hello from mypackage!");
    }
}
```

### Using the Package

```
import mypackage.Hello;

public class Test {
    public static void main(String[] args) {
        Hello h = new Hello();
        h.show();
    }
}
```

### (c) Explain polymorphism with example.

Polymorphism is a fundamental **Object-Oriented Programming (OOP) concept** in Java, which means **"many forms"**.

It allows an **object, method, or operator** to behave differently in **different contexts**.

In Java, polymorphism enables:

- A **single interface** to represent **different types of objects**
- Methods or objects to perform **different behaviors** depending on context

Polymorphism enhances **flexibility, maintainability, and reusability** in Java programs.

---

# Types of Polymorphism in Java

Polymorphism in Java is mainly of **two types**:

### A. Compile-time Polymorphism (Static Polymorphism)

- Occurs **at compile time**.
- Achieved by **method overloading** or **operator overloading**.
- The **method called** is determined during compilation.

**Characteristics:**

1. Same method name with **different parameters** (number or type).
2. Return type can be same or different.
3. Improves readability and reusability.

---

### B. Runtime Polymorphism (Dynamic Polymorphism)

- Occurs **at runtime**.
- Achieved by **method overriding** (subclass provides a specific implementation of a parent class method).
- The **method called** is determined at runtime based on the **object type**.

**Characteristics:**

1. Requires **inheritance**.
2. Uses **overridden methods**.
3. Supports **dynamic method dispatch**.

---

# Advantages of Polymorphism

1. **Code Reusability:** Same method or operator can work in multiple ways.
2. **Flexibility:** New functionality can be added easily without changing existing code.

3. **Maintainability:** Reduces redundancy and simplifies updates.
4. **Supports OOP Principles:** Works closely with **inheritance** and **abstraction**.

---

# Simple Example of Polymorphism

## A. Compile-time Polymorphism (Method Overloading)

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class Test {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(10, 20));       // calls 2-parameter
method
        System.out.println(calc.add(5, 15, 25));    // calls 3-parameter
method
    }
}
```

**Output:**

```
30
45
```

---

## B. Runtime Polymorphism (Method Overriding)

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal a = new Dog();  // reference of parent class, object of
child class
        a.sound();             // calls overridden method at runtime
    }
}
```

**Output:**

```
Dog barks
```

**Explanation:**

- `a.sound()` calls the **Dog class method** even though the reference is of type `Animal`.
- This is **runtime polymorphism**, as the decision is made at runtime.

## Q.4 (a) Explain use of throw in exception handling with example.

In Java, the **`throw` keyword** is used to **explicitly throw an exception**.
It allows a programmer to signal an **exceptional condition** during program execution.

- Syntax:

```
throw new ExceptionType("Error Message");
```

- `throw` is **different from `throws`**:
  - `throw` is used to **throw an exception object**.
  - `throws` is used in the method declaration to **specify exceptions that a method might throw**.

---

# 2. Uses of `throw`

1. • **Explicitly throw exceptions:**
   `throw` is used to manually signal that an error or exceptional condition has occurred during program execution.
2. • **Validate input data:**
   Methods can use `throw` to check input values and stop execution if the data is invalid.
3. • **Trigger built-in exceptions:**
   Common Java exceptions like `ArithmeticException`, `NullPointerException`, or `ArrayIndexOutOfBoundsException` can be explicitly thrown.
4. • **Enforce program rules:**
   `throw` ensures that business or program logic rules are not violated, such as age limits or range checks.
5. • **Throw custom (user-defined) exceptions:**
   Developers can create their own exception classes and use `throw` to signal specific error conditions in their applications.
6. • **Interrupt normal program flow:**
   When an exception is thrown, normal execution stops, and control is transferred to the nearest matching `catch` block.
7. • **Separate error handling from normal logic:**
   Using `throw` allows programmers to keep main logic clean while handling errors in `catch` blocks.

8. • **Provide informative error messages:**
   `throw` can carry messages describing the problem, helping users or developers understand what went wrong.
9. • **Support debugging:**
   By explicitly throwing exceptions, it becomes easier to identify the exact location and cause of errors.
10. • **Improve program robustness:**
    Using `throw` prevents the program from continuing with invalid data or operations, making it more reliable and safe.

---

# 3. Example

```
class TestThrow {
    void checkAge(int age) {
        if(age < 18) {
            throw new ArithmeticException("Age must be 18 or above");
        } else {
            System.out.println("Access granted");
        }
    }

    public static void main(String[] args) {
        TestThrow obj = new TestThrow();
        obj.checkAge(15);  // throws exception
    }
}
```

## (b) Explain creation of different shapes in JAVAFX application?

- JavaFX provides a **rich set of 2D shapes** for creating graphical applications.
- Shapes are **objects of classes** in the `javafx.scene.shape` package.
- You can draw **rectangles, circles, ellipses, lines, polygons**, etc., and add them to a **Pane** or other layout containers.
- Shapes can have properties like **color, stroke, fill, and position**.

---

# 2. Common Shapes in JavaFX

1. **Rectangle**
   - Represents a rectangle with specified **x, y, width, height**.
   - Example:
   - `Rectangle r = new Rectangle(50, 50, 100, 80); // x, y, width, height`
2. **Circle**
   - Represents a circle with **center coordinates and radius**.
   - Example:

- `Circle c = new Circle(150, 150, 50); // centerX, centerY, radius`

3. **Ellipse**
   - Represents an ellipse with **centerX, centerY, radiusX, radiusY**.
   - Example:
   - `Ellipse e = new Ellipse(200, 200, 80, 50);`
4. **Line**
   - Represents a straight line between two points.
   - Example:
   - `Line l = new Line(50, 50, 200, 200); // startX, startY, endX, endY`
5. **Polygon**
   - Represents a closed shape with **multiple points**.
   - Example:
   - `Polygon p = new Polygon(50.0,50.0, 150.0,50.0, 100.0,150.0);`
6. **Polyline**
   - Represents an **open shape** with multiple connected points.
   - Example:
   - `Polyline pl = new Polyline(50.0,50.0, 150.0,50.0, 100.0,150.0);`

(c) Explain Generics classes with example.

A **generic class** in Java is a **class that can operate on objects of various types while providing compile-time type safety**.
Instead of writing separate classes for different data types, a single generic class can work with multiple types by using **type parameters**.
Generics help in creating **reusable, type-safe, and flexible code**. They are widely used in Java, especially in **collections like ArrayList, HashMap, etc.**

**Key idea:** "Write once, use for any reference type."

# . Features

1. **Type Parameter:** A placeholder (like `T`) that represents a data type.
2. **Compile-time Type Checking:** Ensures errors are caught during compilation rather than at runtime.
3. **Code Reusability:** The same class can handle different types without duplicating code.
4. **Type Safety:** Prevents storing the wrong type of object in a class instance.
5. **Flexibility:** Can be used with any **reference type** (`Integer`, `String`, `Double`, etc.).
6. **Widely Used in Collections:** Java Collections like `ArrayList<T>` and `HashMap<K,V>` use generics.

# Syntax of Generic Class

```
class ClassName<T> {
    T data;  // generic type variable
```

```
    void setData(T data) {
        this.data = data;
    }

    T getData() {
        return data;
    }
}
```

- `T` is a **type parameter**.
- Can also use letters like `E` (element), `K` (key), `V` (value) for different purposes.
- When creating objects, `T` is replaced with the **actual data type**.

**Example:**

```
// Generic class
class Box<T> {
    private T data;

    void setData(T data) {
        this.data = data;
    }

    T getData() {
        return data;
    }
}

public class TestGenerics {
    public static void main(String[] args) {
        // Integer type
        Box<Integer> intBox = new Box<>();
        intBox.setData(123);
        System.out.println("Integer Value: " + intBox.getData());

        // String type
        Box<String> strBox = new Box<>();
        strBox.setData("Hello Generics");
        System.out.println("String Value: " + strBox.getData());
    }}
```

**Output:**

```
Integer Value: 123
String Value: Hello Generics
```

# Advantages of Generic Classes

1. **Type Safety:** Ensures only the specified type can be used, reducing runtime errors.
2. **Code Reusability:** One generic class can handle multiple data types without rewriting.
3. **Readability:** Code is cleaner and easier to understand, without type casting.
4. **Compile-time Checking:** Errors are detected at compile time, improving program reliability.
5. **Flexibility:** Can work with any reference type and can be extended to methods and interfaces.
6. **Widely Applicable:** Used extensively in collections and APIs for safer and cleaner code.

OR

Q.4 (a) Explain difference between throw and throws.

| Sr. No. | Key | throw | throws |
|---|---|---|---|
| 1 | Definition | Throw is a keyword which is used to throw an exception explicitly in the program inside a function or inside a block of code. | Throws is a keyword used in the method signature used to declare an exception which might get thrown by the function while executing the code. |
| 2 | Internal implementation | Internally throw is implemented as it is allowed to throw only single exception at a time i.e we cannot throw multiple exception with throw keyword. | On other hand we can declare multiple exceptions with throws keyword that could get thrown by the function where throws keyword is used. |
| 3 | Type of exception | With throw keyword we can propagate only unchecked exception i.e checked exception cannot be propagated using throw. | On other hand with throws keyword both checked and unchecked exceptions can be declared and for the propagation checked exception must use throws keyword followed by specific exception class name. |
| 4 | Syntax | Syntax wise throw keyword is followed by the instance variable. | On other hand syntax wise throws keyword is followed by exception class names. |

(b) Write programs to deal with MouseEvents.

```
import java.awt.*;
import java.awt.event.*;

public class MouseEventDemo extends Frame implements MouseListener {

   String msg = "";
   int x = 20, y = 50;

   public MouseEventDemo() {
      addMouseListener(this);
      setSize(400, 300);
      setVisible(true);
   }

   public void paint(Graphics g) {
      g.drawString(msg, x, y);
   }

   public void mouseClicked(MouseEvent e) {
      msg = "Mouse Clicked";
      repaint();
   }

   public void mousePressed(MouseEvent e) {
      msg = "Mouse Pressed";
      repaint();
```

```
    }

    public void mouseReleased(MouseEvent e) {
        msg = "Mouse Released";
        repaint();
    }

    public void mouseEntered(MouseEvent e) {
        msg = "Mouse Entered Window";
        repaint();
    }

    public void mouseExited(MouseEvent e) {
        msg = "Mouse Exited Window";
        repaint();
    }

    public static void main(String[] args) {
        new MouseEventDemo();
    }
}
```

(c) Explain Generics methods with example.

# Definition

A **generic method** in Java is a method that is written with **type parameters**, so the method can operate on **different data types** without being rewritten.
Unlike a generic class, where the type parameter applies to the entire class, a **generic method defines its type parameter within the method itself**.
This allows any normal class to contain one or more generic methods.

Generic methods provide **flexibility, type safety, and reusability**, ensuring that type errors are caught at compile time.

---

## 2. Key Features of Generic Methods

1. The type parameter is written **before the return type**, e.g., `<T>`.
2. A single method can work with **Integer, String, Double, or any object type**.
3. It avoids the need for method overloading for different data types.
4. Type checking happens at **compile time**, making the program safer.
5. They can exist inside **normal classes** as well as **generic classes**.

---

## 3. Syntax of a Generic Method

```
<type-parameter> returnType methodName(type-parameter variable)
```

Example:

```
public <T> void display(T value) {
    System.out.println(value);
}
```

---

# 4. Simple Example of Generic Method

```
class GenericMethodExample {

    // Generic method
    public <T> void printData(T data) {
        System.out.println("Value: " + data);
    }

    public static void main(String[] args) {
        GenericMethodExample obj = new GenericMethodExample();

        obj.printData(100);          // Integer
        obj.printData("Hello");      // String
        obj.printData(45.67);        // Double
    }
}
```

---

## 5. Explanation

- `<T>` is the **type parameter** declared before the return type.
- The same method `printData()` is used to print an integer, a string, and a double.
- No overloading or casting is required.
- Java automatically replaces `T` with the actual data type during method call.

- This improves **code reusability** and **type safety**.

---

## 6. Advantages of Generic Methods

1. **Reusability** – One method works for all data types.
2. **Type Safety** – Catches type mismatch errors during compilation.
3. **Clean Code** – No need to write multiple overloaded versions of the same method.
4. **Flexibility** – Works with classes, arrays, collections, and custom objects.
5. **Better Performance** – Avoids unnecessary type casting.

Q.5 (a) Demonstrate use of the Animation, PathTransition.

**Animation in JavaFX** is a framework used to create motion, visual effects, and changes in UI elements over time.
It allows you to animate properties like position, size, color, rotation, opacity, scale, etc.

---

# Uses of Animation

1. **To move UI components smoothly** across the screen.
2. **To rotate, scale, fade** or apply other visual effects.
3. **To create attention-grabbing UI** (buttons, icons, loaders).
4. **For game development** (moving characters, objects).
5. **For transitions between scenes/pages** in applications.
6. **To improve user experience** by adding life to static elements.

---

# Example

This animation moves a rectangle from left to right.

```
import javafx.animation.TranslateTransition;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class SimpleAnimation extends Application {
    public void start(Stage stage) {
        Rectangle rect = new Rectangle(50, 50, Color.GREEN);
```

```
        TranslateTransition tt = new
TranslateTransition(Duration.seconds(2), rect);
        tt.setByX(200);    // move right
        tt.setCycleCount(TranslateTransition.INDEFINITE);
        tt.setAutoReverse(true);
        tt.play();

        Pane root = new Pane(rect);
        stage.setScene(new Scene(root, 300, 200));
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

---

# 2. PathTransition – Definition

**PathTransition** is a type of animation in JavaFX where a node (shape) moves along a predefined path such as a line, circle, curve, etc.

---

# Uses of PathTransition

1. **To move an object along a custom shape path** (circle, zig-zag, curve).
2. **To simulate real-world movement**, like cars on a road or planets orbiting.
3. **To create UI decorations**, such as scrolling banners or moving icons.
4. **Useful in educational apps**, like showing motion along a graph or diagram.
5. **Creating game animations**, e.g., enemy movement patterns.
6. **To add creative effects** for logos or introductory animations.

---

# Example – PathTransition

This example moves a circle along a line path.

```
import javafx.animation.PathTransition;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.stage.Stage;
import javafx.util.Duration;

public class SimplePathTransition extends Application {
```

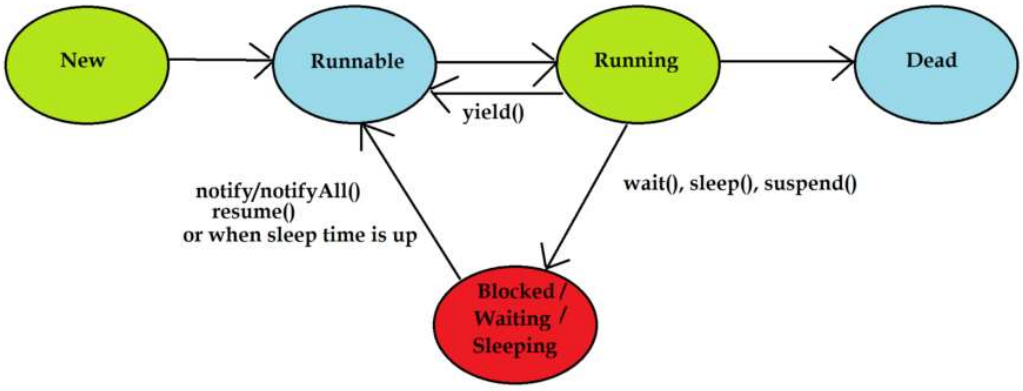
```
    public void start(Stage stage) {
        Circle circle = new Circle(15, Color.RED);
        Line path = new Line(50, 100, 250, 100);  // movement path

        PathTransition pt = new PathTransition(Duration.seconds(3), path,
circle);
        pt.setCycleCount(PathTransition.INDEFINITE);
        pt.setAutoReverse(true);
        pt.play();

        Pane root = new Pane(circle, path);
        stage.setScene(new Scene(root, 300, 200));
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

(b) Describe the life cycle of a thread object.




Thread Lifecycle using Thread states

# Thread Life Cycle

A **thread life cycle** represents all the stages through which a thread passes from its creation to the end of execution. In Java, a thread is an independent path of execution, and its life cycle is controlled by the JVM.

The major states are:
**New → Runnable → Running → Blocked/Waiting/Sleeping → Dead**

Below is the complete explanation of each state:

---

# 1. New State

- When a thread object is created using the `Thread` class or implementing `Runnable`, it enters the **New** state.
- The thread is not yet scheduled for running.

**Example:**

```
Thread t = new Thread();
```

---

# 2. Runnable State

- When the `start()` method is called, the thread moves to the **Runnable** state.
- In this state, the thread is ready to run, but the CPU has not yet assigned time to it.
- Multiple runnable threads wait in the ready queue.

**Example:**

```
t.start();
```

---

# 3. Running State

- When the thread scheduler selects a thread from the runnable pool, it enters the **Running** state.
- The thread actually begins executing the `run()` method.
- At any time, the thread may lose the CPU when:
    - it completes its time slice
    - another high-priority thread arrives
    - it calls `yield()`

---

# 4. Blocked / Waiting / Sleeping State

A thread enters this state when it is temporarily inactive due to:

### (a) Blocked

- When a thread tries to access a resource locked by another thread.
- Occurs during **synchronized** operations.

### (b) Waiting

- When a thread waits for another thread to perform an action.
- Methods: `wait()`, `join()`

### (c) Sleeping

- When a thread is forcefully paused for a certain time.
- Method: `sleep(time)`

**A thread leaves this state when:**

- `notify()` / `notifyAll()` is called
- sleep time is over
- resource lock is released

---

# 5. Dead (Terminated) State

- A thread enters the **Dead** state when its `run()` method finishes execution.
- It can also enter this state due to an **uncaught exception**.
- Once dead, a thread **cannot be started again**.

(c) Explain use of Linked List collection class with example.

The **LinkedList** class in Java is part of the **java.util** package and implements both the **List** and **Deque (Double-Ended Queue)** interfaces. It is based on a **doubly linked list** data structure, meaning every element (node) holds references to both the previous and next nodes.

Unlike an ArrayList, a LinkedList does **not** store elements in continuous memory. Each node is connected using links, which makes insertion and deletion operations much faster especially in the middle of the list.

---

# Key Features / Uses of LinkedList

### 1. Efficient insertion and deletion

- Adding or removing elements from the beginning, end, or middle is very fast (O(1) for ends, O(n) for middle).
- No shifting of elements like in ArrayList.

### 2. Acts as List, Queue, and Deque

- Works as a **List**: can store data in ordered form.
- Works as a **Queue**: supports FIFO operations like `offer()`, `poll()`.
- Works as a **Deque**: can add/remove from both ends (`addFirst()`, `addLast()`).

### 3. Allows duplicate elements

- Just like ArrayList, LinkedList also supports duplicates.

### 4. Good choice when frequent insertions/deletions occur

- If your program frequently adds/removes items from the middle, LinkedList is more suitable.

### 5. Maintains insertion order

- Elements are kept in the order they were inserted.

---

# Simple Example of LinkedList

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> fruits = new LinkedList<>();

        // Adding elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");

        // Adding at beginning and end
        fruits.addFirst("Orange");
        fruits.addLast("Grapes");

        // Removing an element
        fruits.remove("Banana");

        // Displaying the list
        System.out.println("LinkedList: " + fruits);
    }
}
```

### Output:

```
LinkedList: [Orange, Apple, Mango, Grapes]
```

---

# Advantages of LinkedList

### 1. Fast insertion and deletion

- Adding or removing elements anywhere in the list is faster because no shifting of elements is required.

### 2. Dynamic size

- The size of a LinkedList grows or shrinks automatically at runtime, so you don't need to define a fixed size.

### 3. Efficient use of memory

- Memory is allocated only when a new node is created. No wasted space like arrays.

### 4. Good for queues and stacks

- Because LinkedList supports insertion/removal from both ends, it is ideal for implementing **queue**, **stack**, and **deque**.

### 5. Maintains insertion order

- LinkedList preserves the order in which elements are added.

### 6. No memory shifting

- When removing or adding elements in the middle, only node links change—not the entire structure.

### 7. Supports both forward and backward traversal

- Since it is a **doubly linked list**, you can move in both directions (previous and next).

OR

## Q.5 (a) Create a radio button using the RadioButton class and group radio buttons using a ToggleGroup.

In JavaFX, **RadioButton** is used when you want the user to select **only one option** from a group.
To ensure that only one RadioButton is selected at a time, we use **ToggleGroup**.
All RadioButtons added inside the same ToggleGroup behave like a group—selecting one automatically deselects the others.

```
import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.RadioButton;

import javafx.scene.control.ToggleGroup;
```

```
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class RadioButtonExample extends Application {

   public void start(Stage stage) {

      // Create RadioButtons
      RadioButton r1 = new RadioButton("Option 1");
      RadioButton r2 = new RadioButton("Option 2");
      RadioButton r3 = new RadioButton("Option 3");

      // Create a ToggleGroup to group radio buttons
      ToggleGroup group = new ToggleGroup();
      r1.setToggleGroup(group);
      r2.setToggleGroup(group);
      r3.setToggleGroup(group);

      // Layout
      VBox root = new VBox(10, r1, r2, r3);

      // Create Scene & Stage
      Scene scene = new Scene(root, 300, 200);
      stage.setTitle("RadioButton Example");
      stage.setScene(scene);
      stage.show();
   }
   public static void main(String[] args) {
      launch(args); }    }
```

(b) Explain runnable interface.

In Java, the **Runnable interface** is used to create and execute threads. It is a functional interface present in the **java.lang** package and contains only **one abstract method**, named `run()`. Since Java allows only **single inheritance**, using Runnable gives an alternative way to create threads without extending the `Thread` class.

A class that implements `Runnable` must provide the implementation for the **run()** method. This method contains the code that will be executed inside the thread.

Runnable is especially useful when:

- a class already extends another class
- multiple threads need to share the same task
- we want clean separation between the task (Runnable) and the thread (Thread class)

# Key Points of Runnable Interface

### 1. Provides a way to create threads without extending Thread

Implementing Runnable allows you to keep extending some other class, as Java does not support multiple inheritance.

### 2. Contains only one abstract method – run()

This method holds the code that the thread executes.

### 3. Used to share the same task among multiple threads

Multiple Thread objects can execute the same Runnable object.

### 4. Better design practice

Runnable separates the **job (task)** from the **worker (thread)**, making the code more flexible and maintainable.

### 5. Lightweight compared to extending Thread

It avoids inheriting many unnecessary properties of the Thread class.

```
class MyTask implements Runnable {

   public void run() {

      System.out.println("Thread is running...");

   }
```

```
}

public class Test {

   public static void main(String[] args) {

    MyTask task = new MyTask();     // create runnable object

       Thread t = new Thread(task);    // pass it to thread

       t.start();                    // start thread

   }

}
```

**Output:**

```
Thread is running...
```

(c) Explain Sets with examples.

---

# What is a Set in Java?

A **Set** in Java is a part of the **Collection Framework** and represents a group of elements where **duplicate values are not allowed**. It models the real-world mathematical *set* where each element is unique. Set is defined in the **java.util** package and implemented by several classes like:

- **HashSet**
- **LinkedHashSet**
- **TreeSet**

Set does **not maintain index**, meaning elements cannot be accessed by position (like list.get(0)). Instead, the Set focuses on **uniqueness** and **fast searching**.

---

# Key Characteristics of Set

### 1. No Duplicate Elements Allowed

A Set automatically removes duplicates.
If you insert the same element again, it simply ignores it.

### 2. Unordered (in HashSet)

HashSet does not maintain insertion order.
TreeSet maintains **sorted order**.
LinkedHashSet maintains **insertion order**.

### 3. No Index-Based Access

You cannot access elements using indexes (like `set.get(2)`), because Set does not support indexing.

### 4. Efficient Searching

Set provides very fast searching due to hashing (especially HashSet).

### 5. Can Store Only Unique, Non-Duplicate Values

Useful when you want to ensure uniqueness such as:

- storing roll numbers
- storing email IDs
- storing unique product names

---

# Types of Sets in Java (Simple Explanation)

Here are the **longer 5–6 line explanations** for each:

---

# 1. HashSet

HashSet is a commonly used Set implementation in Java that stores elements using a hash table. It does **not maintain any order**, meaning elements appear in random order based on their hash values. HashSet is very fast for operations like **add**, **remove**, and **search** because hashing provides near-constant time performance. It automatically avoids duplicate values and stores only unique elements. It is useful when you only need uniqueness, not ordering.

---

# 2. LinkedHashSet

LinkedHashSet is similar to HashSet but it maintains the **insertion order** of elements. Internally, it uses both a hash table and a **doubly linked list**, which preserves the order of insertion. It is slightly slower than HashSet because maintaining the linked list requires extra

processing. LinkedHashSet still does not allow duplicate elements and offers good performance for lookup operations. It is useful when you want uniqueness with predictable ordering.

---

# 3. TreeSet

TreeSet stores elements in **sorted (ascending) order** using a **Tree (Red-Black Tree)** structure. Because of this, operations like insertion, deletion, and searching are slightly slower (O log n) compared to HashSet. TreeSet does not allow duplicates and automatically arranges elements in a natural or custom order. It is ideal when your application requires both uniqueness and automatic sorting, such as storing sorted numbers or names.

---

# Example 1: HashSet

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> names = new HashSet<>();

        names.add("Nisha");
        names.add("Rita");
        names.add("Nisha");   // duplicate, will be ignored
        names.add("Raj");

        System.out.println(names);
    }
}
```

### Output:

```
[Rita, Nisha, Raj]
```

---

# Example 2: LinkedHashSet

```
import java.util.LinkedHashSet;

public class LinkedHashSetExample {
    public static void main(String[] args) {
        LinkedHashSet<Integer> set = new LinkedHashSet<>();

        set.add(10);
        set.add(20);
        set.add(10); // duplicate ignored
        set.add(30);

        System.out.println(set);
```

```
    }
}
```

### Output:

```
[10, 20, 30]
```

---

# Example 3: TreeSet (Sorted Set)

```
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();

        numbers.add(50);
        numbers.add(10);
        numbers.add(30);

        System.out.println(numbers);  // automatically sorted
    }
}
```

### Output:

```
[10, 30, 50]
```

---